Getting Started with the Intel® Thread Checker and Thread Profiler

Contents

About this Guide 4

System Requirements 5

Hardware Requirements 5
Recommended Hardware Configuration 5
Software Requirements 5
Intel® Compilers 5

Introduction 6

About the Threading Tools 6
The Intel® Thread Checker 6
Thread Profiler 6
Terminology 7

Checking an Application Using the Intel® Thread Checker 8

1. Verify that your Fortran Program Does Not Rely on Static Allocation 8
2. Instrument your Application 8
3. Select a Small but Representative Data Set 9
4. Use the Thread Checker Wizard 9
5. Investigate Errors and Compare Results 10
Related Documentation 12

Profiling a Sample OpenMP* Application Using Thread Profiler 13

1. Build and Link your Application 13
2. Use the Thread Profiler Wizard 13
3. Investigate and Compare Results 14
4. Advice on Performance Issues 16

Appendix A. Building Code for Use with Intel® Thread Checker 17

Binary Instrumentation Mode 17
Binary Instrumentation Options 17
Source Instrumentation Mode 18
Mixed Mode 18

About this Guide

This getting started guide takes you through all the basic steps required to prepare your threaded application for data collection, then generate and analyze data using the Intel® Thread Checker and thread profiler. Additional details on these tools are provided in the product's online help.

Additional details on the Intel compiler are provided in that product's documentation.

System Requirements

The following hardware and software components are required to successfully analyze your threaded application using the Intel® Thread Checker and thread profiler.

For the latest requirements, be sure to check the product Release Notes.

Hardware Requirements

- Intel® Pentium® III or Intel® Pentium® III Xeon™ processor-based system or later
- 128 MB of RAM[+]
- 100 MB of disk space

Recommended Hardware Configuration

- Intel® Pentium® 4 or Intel® Xeon™ processor-based system or later
- 256 MB of RAM[+]

[+]Intel® Thread Checker may use up to 20x the amount of memory that your base application needs to run. Therefore, you should be sure that that your development system has ample RAM.

Software Requirements

- Any 32-bit version of Windows* 2000 or Windows* XP excluding the Home Editions
- VTune™ Performance Analyzer version 6.1 or higher

Intel® Compilers

For the thread profiler, one of the following Intel compilers is required. For the Intel® Thread Checker, using one of the following compilers is recommended, and is required to use source instrumentation.

- Intel® C++ Compiler for Windows, version 7.0 or higher
- Intel® Fortran Compiler for Windows, version 7.0 or higher

The Intel® Thread Checker "instruments" source code and executables to find errors. For source instrumentation, the Intel C++ or Fortran compilers specified in the requirements section must be used. Source instrumentation enables Thread Checker to provide more details about threading errors.

To learn how to obtain the Intel® Compilers, see the Intel Software Development Products website at: http://www.intel.com/software/products/. Be sure to see the compiler's documentation for more information about the compilers.

Introduction

About the Threading Tools

The Intel® Thread Checker and thread profiler help you produce correctly threaded software and to improve the threaded software's performance. This section introduces the tools. The rest of this guide shows you how to use the tools to gather and analyze data about your threaded software.

The Intel® Thread Checker

Use the Intel® Thread Checker with the VTune™ Performance Environment to validate the correctness of multithreaded programs using the Microsoft* Windows* 32-bit threading or OpenMP*. Thread Checker identifies errors induced by the non-deterministic scheduling of thread execution.

Multiple threads can create a situation where the same memory location (variable) is accessed simultaneously by more than one thread. This is a type of race condition known as a storage conflict or a data race. Data races can cause erratic program behavior and lead to indeterminate results.

Deadlock is another type of race condition in which a thread waits for a resource that it can never acquire or an event that will never happen. Bad locking hierarchies (where two threads hold separate locks and need to obtain the lock held by the other) can also deadlock threads. Thread Checker identifies the following errors and potential issues such as:

- data races
- deadlocks
- stalled threads
- lost signals
- abandoned locks

Because Thread Checker keeps track of all memory accesses, there is significant overhead in using it. However, you can drastically reduce the overhead by reducing your overall workload.

NOTE

Reduction of workload is a critical, required step before running Thread Checker. The workload reduction is similar to what might be done to reduce the data set when setting up an interactive debugging session.

Thread Profiler

The thread profiler is a VTune analyzer data collector. Use thread profiler to locate bottlenecks that are limiting the parallel performance of your OpenMP* software.

With the thread profiler, you can:

- Compare the performance impact of using different configuration options when your program is run, such as the thread scheduling method (dynamic or not) or the number(s) of threads used to run your application.
- Determine if load imbalance is hurting parallel efficiency.

- Locate synchronization constructs with excessive lock contention.
- Estimate the total run time that you would get if more processors were available. This enables you to estimate how well your program's performance could scale on machines that have more processors.

Terminology

The Intel® Thread Checker and thread profiler enable you to create Activities in the VTune™ Performance Environment. Activities are at the core of the data-collection process in the VTune analyzer. Within an Activity, you can specify the types of performance data you wish to collect. For each type of performance data, you need to configure the appropriate data collector and define the application/module profiles. These profiles contain information about the application to execute and the modules to analyze.

Running an Activity is analogous to running an experiment. When you run an Activity, the data collectors collect performance data and save it in the Activity results. The results can be viewed later, without having to rerun the Activity. You can also drag and drop different Activity results into a view to compare data from different experiments.

This guide walks you through the creation of an Activity using the Intel® Thread Checker and the thread profiler.

Checking an Application Using the Intel® Thread Checker

Follow the instructions in this section to use the Intel® Thread Checker to check your multi-threaded code for errors and potential problems.

1. Verify that your Fortran Program Does Not Rely on Static Allocation

If you are not using Fortran, skip to step 2.

If you are using Fortran, you need to verify that your program does not rely on static allocation of local variables for correct execution. Most Fortran compilers provide an option to make all local variables static that is, using the `-Qsave` option of the Intel Fortran compiler. This is equivalent to declaring variables with the `SAVE` attribute on variables, placing them in a `SAVE` statement, or initializing them with a `DATA` statement. Subprograms with static variables are not thread-safe for two reasons: static variables are shared among threads and static variables maintain state between invocations.

To test whether your program relies on static allocation:

- Compile your program with the `-Qauto` option of the Intel Fortran compiler to place local variables on the stack.
- If your program gives correct results, proceed to step 2. If your program gives incorrect results, this generally indicates that your code has subprograms that maintain state across invocations.
- Find the subprograms that maintain state. Identify the variables in these functions whose values must be saved across invocations.
  For example, the seed value of a random number generator is often saved across invocations.
- Give these variables the `SAVE` attribute or place them in a `SAVE` statement, for example, using one of the following statements:
  ```
  INTEGER, SAVE :: static_variable
  ```
  or:
  ```
  INTEGER static_variable
  SAVE static_variable
  ```
- Recompile with the `-Qauto` option and verify that the program gives correct results. Proceed to step 2 knowing that static variables are shared among threads. The Intel® Thread Checker may report data races on these variables that you will have to fix.

2. Instrument your Application

Collecting data with Thread Checker requires instrumentation. With instrumentation, calls to the Thread Checker library are inserted into your code to record memory accesses and calls to threading and synchronization routines. You can use either source or binary instrumentation methods or a combination of both, depending on the needs of your program.

If you are using OpenMP*, use source instrumentation as described in

Appendix A Otherwise, you can use binary instrumentation to get started.

To do binary instrumentation:

Use a debug build that includes symbols (`-Zi` or `/ZI`) and disables optimization (`-Od`) of your software and link it with the `/fixed:no` option. Also, be sure to build with thread-safe run-time libraries by using the `–MDd` or –MTd option. For example, to build `test.exe` with the Microsoft* Compiler, use the following:

```
cl -MDd -Od -Zi test.cpp /link /fixed:no
```

3. Select a Small but Representative Data Set

CAUTION

Thread Checker instrumentation can drastically increase the runtime and memory requirements of your application so it is very important that you choose a small but representative data set rather than a large, production-scale data set. Data sets with runtimes of a few seconds are ideal.

Because Thread Checker keeps track of all memory accesses, there is significant overhead in using it. However, you can drastically reduce the overhead by reducing your overall workload.

To reduce your workload, use one or more of the following strategies:

- **Reduce input data sets sizes.**
  For example, if you are analyzing an image, use a 10x10 pixel image, not a 2000x1000 pixel image.
- **Reduce loop iterations.**
  Usually a few iterations uncover all the problems in a loop.
- **Reduce update rates.**
  For example, limit the number of display updates from thirty to three per second.

The workloads do not have to be realistic. They just have to exercise the relevant sections of the multithreaded program. This is similar to the smaller workloads that are typically used in serial debugging.

4. Use the Thread Checker Wizard

The wizard enables you to create a new project and a new Intel® Thread Checker Activity that gathers and analyzes information on multi-threading correctness. The wizard prompts you to enter values only for the basic parameters and uses default values for others.

To launch the Intel® Thread Checker wizard:

1. Start the VTune™ Performance Analyzer.
2. Click New Project  in the main toolbar to open the New Project dialog box.
3. In the **Category** drop-down box, choose **Threading Wizards**.
4. Select **Intel® Thread Checker Wizard**  .
5. Click **OK**.
6. Use the Intel® Thread Checker Wizard to configure the Activity with the application to run, the working directory, and the command-line

arguments to use, if any. For all other options, you can use default values.

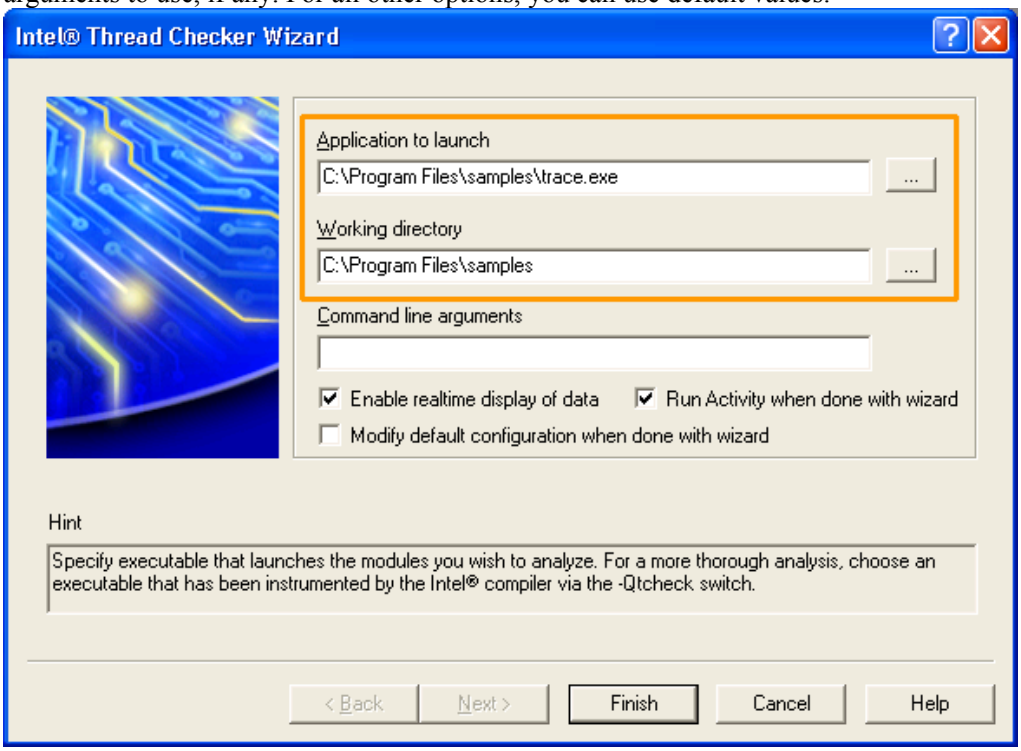


Figure 1: Intel® Thread Checker Wizard.

7. Click **Finish** to run your application and collect results.

Thread Checker instruments your application before execution. Remember that instrumentation expands runtime and memory requirements so be sure to select a small but representative data set as described in Step 3.

5. Investigate Errors and Compare Results

If you have successfully run the Thread Checker Wizard, you should see Activity results in the Project Navigator pane, and messages about errors or warnings in the Diagnostics list and Graphical Summary panes. Congratulations! You are now ready to analyze errors, if any, in your threaded application.

The Thread Checker is particularly good at finding data races, in which multiple threads access a shared variable without synchronization and at least one thread is attempting to update the variable. Some of the things you can do to better understand the errors reported by Thread Checker include:

1. Click the **Severity** column heading to sort errors according to severity, placing the most serious errors at the top of your list.

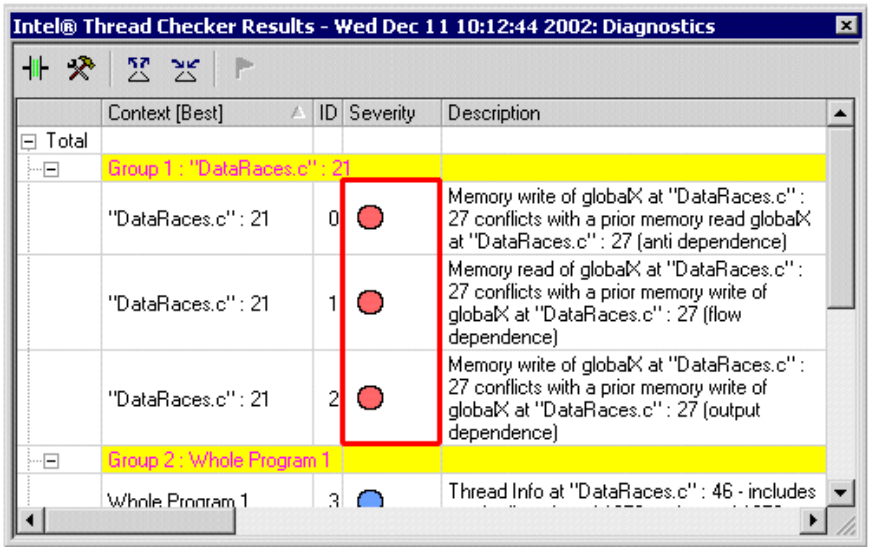


Figure 2: Diagnostics list, with errors grouped by Severity.

2. Double-click an error to drill-down to source views to see where errors are in your code. Look for the variables that are accessed in these lines of the program. At least one of these accesses is a write to the variable. This is the access that can give you insight into the error. Tab through **Context**, **Definition**, **1st** and **2nd Access** views to gain different perspectives on the source of the issues.

   Context View. For OpenMP* code, the Context view takes you to the parallel region where the current error exists in the source code, to provide the "context" of the error. For code that does not use OpenMP, the Context view shows the function where the error is located.

   Definition View. Typically an error results when two different threads are attempting access the same piece of memory at once. The Definition view attempts to take you to the place where the variable was declared or allocated.

   1st Access and 2nd Access Views. The 1st Access location is the line that the first thread was executing at when the conflict occurred. The 2nd Access is the location that the second thread was executing at when the conflict occurred.

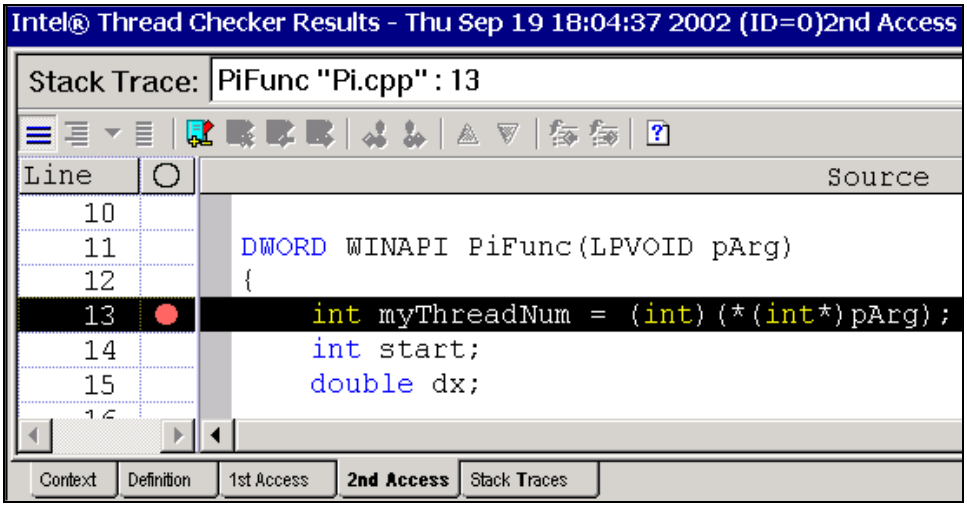
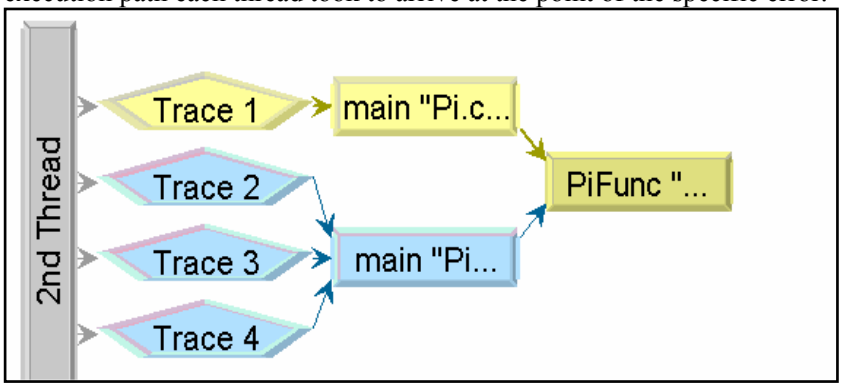


Figure 3: 2nd Access view is one of several source views.

3. Click the **Stack Traces** tab at the bottom of the view to understand the execution path each thread took to arrive at the point of the specific error.



Tips

- If the counts in the error view are large (100s or greater) then you could reduce your workload and get the same results from Thread Checker.
- If you want to see more details about the error, use source instrumentation to create an additional Activity. See Source Instrumentation Mode for instructions.

As you develop solutions to the errors found by Intel® Thread Checker, the modified code can be recompiled and run through Thread Checker again. If there are many errors found, you may wish to only address a few at a time before passing the code through the tool again to see if your solutions were correct and did not introduce any new errors. Depending on your work, it may take several iterations of fixing and checking to eliminate all errors. If you are using OpenMP* threads, you can use the thread profiler. If you are not using OpenMP, proceed to Appendix A. Building Code for Use with Intel® Thread Checker.

Related Documentation

The online help included with the Thread Checker includes more complete details on advanced Thread Checker configuration options and Thread Checker views.

Profiling a Sample OpenMP Application Using Thread Profiler*

This section offers a general model for using the thread profiler to help you identify and locate bottlenecks that may be limiting the parallel performance of your OpenMP* threaded software.

1. Build and Link your Application

Before you begin, you need to build your application to use the instrumented OpenMP* runtime library. You can do this by compiling and linking with the `-Qopenmp_profile` option of the Intel® C++ or Fortran Compiler, v7.0 or higher. See the online help for details. The instrumented runtime collects performance statistics for your OpenMP application.

2. Use the Thread Profiler Wizard

The **Thread Profiler Wizard** enables you to create a new project and Activity with the thread profiler collector that gathers and analyzes an OpenMP* application.

To launch the Thread Profiler Wizard:

1. Start the VTune™ Performance Analyzer.
2. Click New Project  in the main toolbar to open the **New Project** dialog box.
3. From the **Category** drop-down box, select **Threading Wizards**.
4. Select  **Thread Profiler Wizard**.
5. Click **OK**.
6. Select an OpenMP*-enabled executable.
   After you specify an executable, the working directory defaults to the directory containing the executable you selected.
7. Supply any command line arguments needed.
8. Click **Finish** to close the dialog box and create the new project and Activity that uses the executable, directory, and command line options you specified.

The thread profiler creates an Activity result containing data about factors that may limit the parallel performance of your OpenMP application.

Note that unlike the Intel® Thread Checker, the thread profiler requires multiple processors (either physical processors or processors with Hyper-Threading Technology) to collect meaningful data. Also, unlike Thread Checker, data should be collected using a production-sized test problem to obtain an accurate profile.

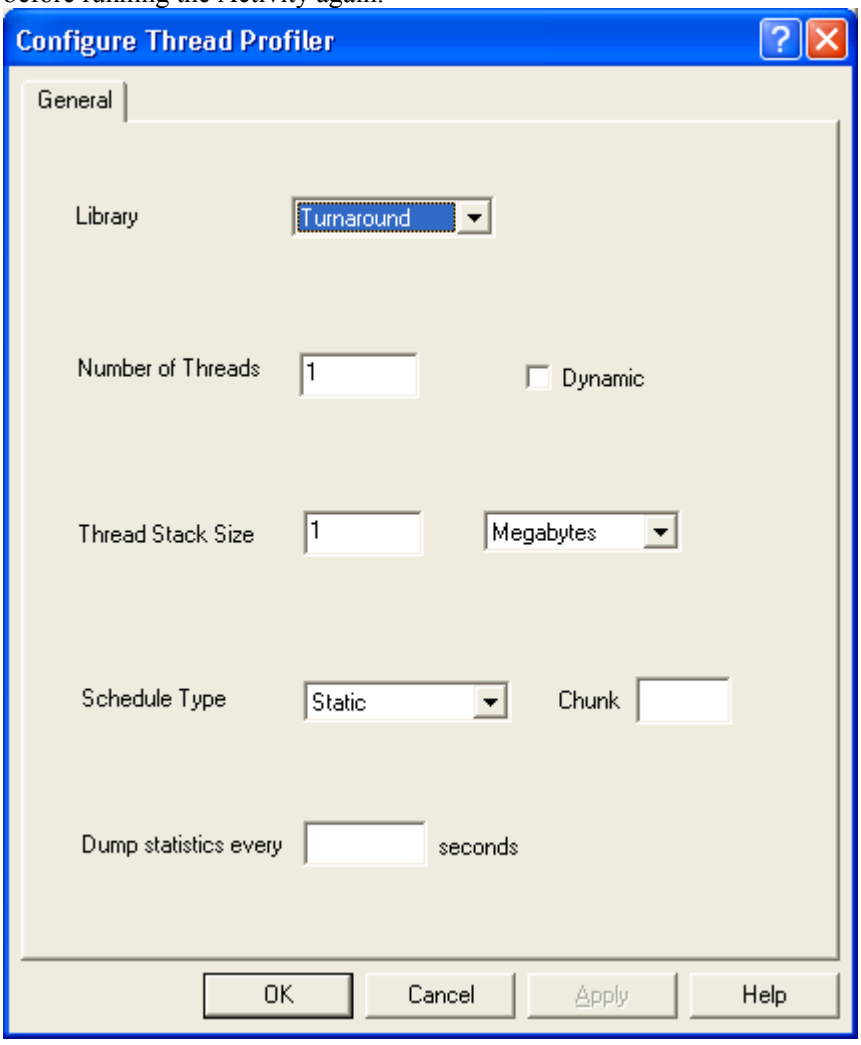
3. Investigate and Compare Results

If you have successfully run the Thread Profiler Wizard, you should see Activity results in the thread profiler viewer. You are now ready to identify and locate bottlenecks that are limiting the parallel performance of your software.

a) Click the tabs at the bottom of the view to navigate between **Summary**, **Threads**, **Regions**, and **Data** Views.

b) Click the histogram bars to display and study the legend to understand where time was spent by your application.

c) Choose **Configure** > **Modify** > **Modify Selected Activity** to open the **Configure Thread Profiler** dialog box and change various runtime parameters such as runtime scheduling and the number of threads before running the Activity again.



d) Drag and drop Activity results from the Project Navigator onto the thread profiler viewer (bar graph) to compare results. After you drop the Activity result additional bars appear in the graph representing the additional Activity result. Results are shown side-by-side with the bars that were already present

from the previous Activity results.

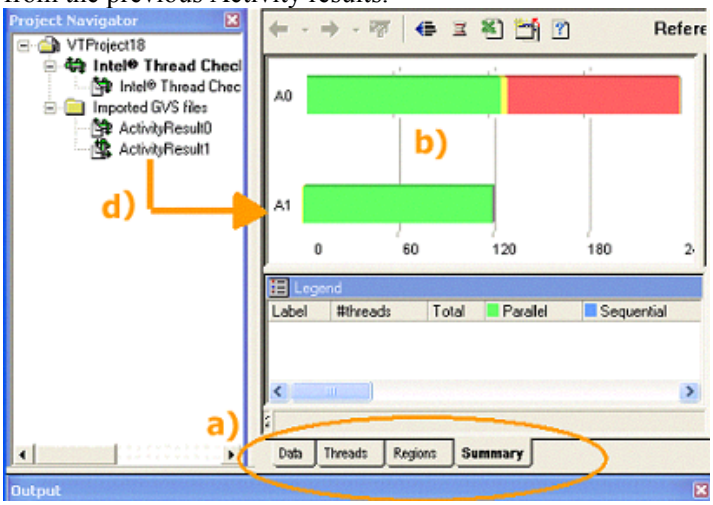


Figure 4: Summary view.

e) Estimate potential speedup by studying the projections in the speedup plot. Applications that are parallelized properly are more likely to scale well (as in Figure 5 than applications that are not well parallelized (as in Figure 6). You can see in Figure 5 that almost all time is parallel time and as more processors become available the speedup graph projects nearly ideal speedup. In contrast, Figure 6 shows a projected speedup (the red line) of only a bit more than 2x when four processors are used because of lock imbalance.

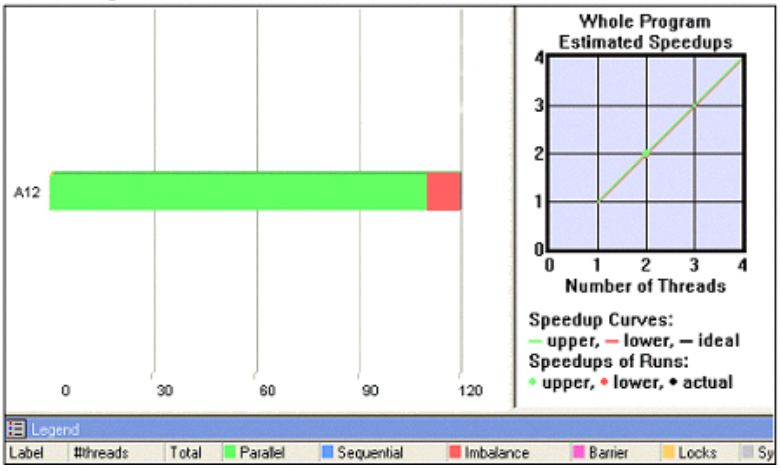


Figure 5: An Activity result for software with good parallelization shows little opportunity for speedup.

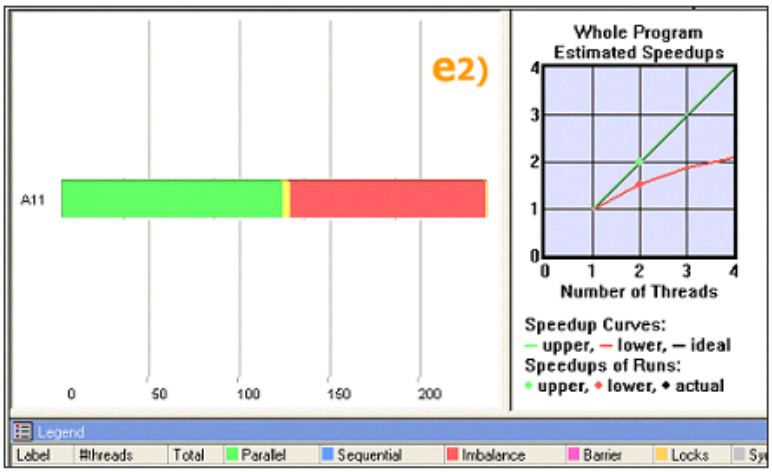


Figure 6: An Activity result for software with poor parallelization shows much opportunity for speedup.

4. Advice on Performance Issues

You are now ready to review advice related to specific performance issues. See the *Thread Profiler Advice* section of the online help.

Appendix A. Building Code for Use with Intel® Thread Checker

There are two ways to prepare your application for data collection with the Intel® Thread Checker:

- binary instrumentation mode
- source instrumentation mode

You can use either method or a mix of both, depending on the needs of your program. These methods and their usage are described in detail in the following sections. For OpenMP-threaded programs, you must use source instrumentation mode.

Binary Instrumentation Mode

Binary instrumentation can be used with software built with the Intel® Compilers or the Microsoft* Visual* C++ Compiler. Even if your software is built with the Intel® Compilers, you may want to use binary instrumentation to quickly get started with only a re-link, and not a whole re-compile.

Because of limitations on what can be done with binary instrumentation, fewer details about any found errors will be available than when source instrumentation is used.

Binary instrumentation is recommended for use in the following situations:

- To quickly get started using the Intel Thread Checker.
- When you do not have access to an appropriate Intel® compiler.
- When you do not want to completely re-build your application, for example, because it might take several hours to do so.
- When you do not have source code.

Binary Instrumentation Options

Consider using the following options to control the way binary instrumentation for the Intel® Thread Checker is performed.

- Build with thread-safe libraries using one of: `-MD -MDd -MT -MTd` compilers options. This applies if you are using the Intel Compiler or if you are using Microsoft* Visual* C++. In Microsoft* Visual* C++ version 6 this can be found under the menu **Project>Settings> C/C++ >Code Generation** by selecting one of the Multi-threaded DLL options.
- Build with symbols and line numbers, using one of: `-Zi -ZI -Z7`. These options enable Thread Checker to show you the source code corresponding to threading errors.
- Disable optimization with `-Od`. Turning on optimization will not speed Thread Checker and may actually slow it down.
- Link so that the executable can be re-located with the `/fixed:no` option. Most .DLL files default to linking `/fixed:no`, but most .EXE files need to have this option specified.

Source Instrumentation Mode

Source instrumentation provides more information than binary instrumentation. It is also required if you want to analyze OpenMP*.

To do source instrumentation:

Use the `-Qtcheck` compiler option and the `/fixed:no` linker option when building your application. Additionally, you should use binary instrumentation options described above.

NOTE

When using source or binary instrumentation, operation of the Thread Checker causes additional debugging information to be embedded into your object (.obj) files. That information is merged from there into the .exe and .dll files, causing file size to grow. If this causes problems in your debugger, contact Intel's technical support for assistance.

Mixed Mode

You can operate with a mixed mode where source instrumentation is used on some files while binary instrumentation is used on others. This mode may be useful in order to get more detailed information on specific critical files while minimizing the amount of output generated from other files and saving the time that would otherwise be needed to recompile everything.